Semantic Structure: a Basis for Software Architecture

Robb D. Nebbe
Software Composition Group
Institut für Informatik und angewandte Mathematik
Universität Bern, Neubrückstrasse 10
CH 3012 Bern
nebbe@iam.unibe.ch

Introduction

Software architecture plays an important role in both the development and evolution of software systems. Understanding this role require both understanding how software systems are structured and more importantly what this structure tells us. Just knowing the structure is of little value unless we also understand its implications. It would be analogous to receiving an important message in a language we cannot read.

An architecture is based on a choice of components and connectors. This choice fundamentally determines what the structure will be and what this structure tells us about the system. Furthermore, I restrict my use of the term architecture to the semantic structure of an application, a static notion of software architecture.

I believe there are many valid notions of software architecture that exist at different levels of abstraction[1]. What is important is to understand how they are related structurally and what the consequences of this structure are on the software system. This allows us to asses how well a particular notion of software architecture is suited to answering our questions about a software system. In this sense the value of any notion of software architecture is relative to what we want to know about the software system.

This introduces the idea of semantic relevance; when we want to understand what a software system does or we wish to modify it to do something else we are interested in its semantics. If the choice of components and connectors is not closely related to some semantic concept then it undermines the ability of such a structural notion to help us understand the software system.

For example the module structure, in general, tells us very little about what a system can do and how it may evolve. Such is the case with packages in Ada and source files in C++. What they do tell us is how the compiler will search through the source code but they have no guaranteed semantic relevance. A good programmer may adopt certain conventions that communicate a higher-level view of the software system but this is in no way guaranteed by the language.

I suggest that there is a lowest or base-level notion of software architecture that reflects the *semantic structure* of a software system. Other kinds of structure exist in a software system but I do not consider them architectural in nature. Furthermore, all notions of software architecture, whatever their level of abstraction, must be understandable in relationship to such a base-level architecture. This guarantees that they are stable with respect to the semantics, i.e. if the semantics of the software system does not change then neither will the architecture.

I start by introducing my ideas on such a base-level architecture. These are embodied in two complementary notions: domain and situation architectures; both of which are static. I then discuss several architectural styles as defined by Shaw and Garlan [3] and their relationship to domain and situation architectures. I then show how domain and situation architecture help to understand the structure of design patterns and frameworks.

These notions have been applied in the context of the FAMOOS Esprit project. Furthermore, they are based in part on experience with the case stud-

1. An architecture need not be at a higher level of abstraction as suggested in [2]. It is sufficient that is separates the semantic structure from other information.

ies provided by the industrial partners. The other influence is that of formalizing the semantics of programs in terms of an algebra; this implies the use of axiomatic semantics rather than the more conventional operational semantics.

1 Domain and Situation Architectures

The hypothesis behind these notions of software architecture is that is a base-level architecture below which we lose sight of the semantics. Accordingly, the ideas behind a domain and situation architecture are based on the structure of an algebra and a situation expressed in this algebra.

I propose two complementary notions of software architecture[1]. They represent a separation of concerns relating to the distinct nature of the information encompassed by each kind of architecture. Moreover, it is through an understanding of their relationship that we understand what this notion of architecture can tell us about the system.

The initial distinction is between classes and instances. We will use the term *domain* when talking about systems of classes and *situation* when talking about systems of instances.

1.1 Domain Architectures

- A *domain architecture* consists of a set of classes (components) that are connected by the potential relationships (connectors) between these classes as expressed by its set of dependencies.

The term domain reflects the fact that this notion of architecture tells you how the model of your problem domain is structured.

Relationships appear in the parameter types of a class' methods. There are different kinds of relationships and a single relationship may show up across several methods. For example the relationship of an array with the type used to index it is a naming relationship. This is in contrast with the items contained in an array (a component relationship) and the bounds or length of an array (an attribute relationship).

1. remember that architectures are static

The naming relationship is just a convenience to avoid having a set of methods such as first, second, third etc. The difference between an attribute relationship and a component relationship is that an attribute quantifies some property of the whole while a component refers to a part of the whole.

The same relationship may show up in several methods. For example if a class has as a component an address then the type address should show up in a constructor and in a method to access the address.

Relationship may also have different durations. For example when transferring money from one account to another sets up a temporary relationship between two accounts. In contrast opening a new account at a bank creates a relationship that will last longer than the method invocation that accomplishes the action.

It is important to realize that there are different kinds of relationships, that they do not correspond directly to a particular method and that they may have different lifetimes.

1.2 Situation Architectures

- A *situation architecture* consists of a set of instances (components) that are connected by the actual relationships (connectors) between these instances.

The term situation architecture reflects the fact that a situation architecture represents the structure of a particular concrete situation from the problem domain as defined by a configuration of instance and their actual relationships. This is in contrast to a domain architecture, which represents potential relationships.

The relationships are given by the parameters of the methods in contrast with the domain architecture where it is the types of the parameters that tells us the relationship.

The kinds of relationship are the same as in a domain architecture but their nature is different than in a situation architecture. A relationship in a situation architecture is an instance of a general relationship from a domain architecture. The point in time when an situation architecture has mean-

ing is after the static set of instances has been initialized but before execution begins. Only then is it meaningful to talk about the actual set of relationships between these instances.

One issue worth pointing out is what to do with singleton classes, i.e. classes with a single instance. We consider the information to be much more important for the situation architecture since due to their nature (only one instance) they embody information about actual relationships within the software system as opposed to potential relationships.

1.3 Their Relationship

The relationship between a domain and situation architecture is one of instantiation. An situation architecture is an instance of a domain architecture. The situation architecture's components and connectors are instances of the domain architecture's components and connectors. This is virtually identical to the relationship between a database schema and an instance of the database itself.

Although the two kinds of architecture are complementary —both are needed to understand the structure of a software system— they are fundamentally different. One way of thinking of the relationship is that the domain model defines what can and can not happen while an situation model defines, in conjunction with any input, what does and does not happen.

The domain architecture circumscribes the possible relationships between the instances comprising a situation architecture. In essence, a domain architecture is a domain specific language for describing situations. The situation architecture describes a particular instance of its corresponding domain architecture, i.e. a configuration of instances.

This emphasizes the difference between the two kinds of architectures; a domain architecture is general is the sense that it delimits what is possible while an situation architecture is specific because it defines actual relationships between instances.

Each represents a fundamentally different kind of information and any structure in which they are indiscriminately mixed is necessarily incoherent. They are complementary and only together do they allow a full understanding of the structure of a software system as well as its flexibility and potential for evolution.

1.4 Why Dynamic Information is Excluded

A domain architecture is necessarily static since it documents the complete set of potential relationships and all the possible ways in which they may evolve. It captures the possible while excluding the impossible and is the arbitrator of what can and cannot happen in a software system.

In some cases we may dynamically load classes at run time; however, these classes must be consistent with the existing system. In this case, from a theoretical point of view, it is easiest to consider them as always having been part of the domain architecture. Consequently there is no advantage to a dynamic concept of a domain architecture.

In contrast, an situation architecture represents a actual configuration of instances and their existing relationships. During execution this configuration will evolve so a dynamic concept of an situation architecture seems somewhat intuitive. Unfortunately it renders the term situation architecture ambiguous in the sense that there will be huge number of them.

We adopt a more restrictive definition of situation architecture as the static instance structure, i.e. the instance structure that is always present during execution and thus characterizes all run-time configurations of instances. Furthermore, the situation architecture determines which configurations are reachable during execution. Consequently, a situation architecture constrains the possible run-time configurations of the software system in much the same way that the domain architecture constrains the possible situation architectures.

Another reasons for restricting our notion of situation architecture to static information is that this allows us to recover it from the source code. Moreover, the role that it plays in constraining the run-time evolution of a software system is essen-

tial to building an understanding of the system's overall behavior.

2 Related Notions of Structure

Here I will discuss various related notions of architecture, including the pipe and filter, layered and object-oriented architectural styles [2]. The goal is to show to what extent related notions of architecture coexist.

I will also related discuss other concepts that are at least partially architectural in nature, in occurrence design patterns and frameworks. Here, the goal is to relate what is considered as best practice by the OO community with the concepts of a domain and situation architecture.

The goal of the section as a whole is to show that the notions of domain and situation architectures can often be used as a reference to which other notions can be compared.

2.1 Architectural Styles

Interestingly enough many of the architectural styles are radically different in terms of their choice of components and connectors; it is not just a difference in what kinds of components and connectors exist but their very nature. In fact different styles may simultaneously provide useful information about a software system.

I start with the object-oriented style since at first it might seem to be the closest to my notion of a situation architecture but in fact is more of an exception. The other two styles are actually much more closely related to the concepts of domain and situation architectures.

Object-Oriented Style: The object-oriented style is based on instances (components) and method invocations[1] (connectors). The is quite different from the previously defined notion of situation architecture. The choice of components is the same but the radically different choice of connectors leads to very different results.

The difference is fundamental making the two difficult to understand in terms of each other. It must suffice to note the differences. For example the situation architecture is based on the visible relationships between instances and is thus closely related to the axiomatic semantics of the software system. In contrast method invocations clearly tie the object-oriented architectural style to the operational semantics.

For example choosing to implement a stack with a list is not reflected in a situation or domain architecture but it is if we consider method invocations.

Another example is the different kinds of relationships present in a situation architecture. An individual relationship may appear across more than one method. In contrast, method invocation provides only a single kind of connector.

We will see that it is of a different nature than the two following styles. This is because it is not related to the semantic structure of an application but to the algorithmic structure of a method. Under the terminology used in this paper Shaw and Garlands object-oriented architectural style is not architectural in nature.

Pipe and Filter Style: The pipe and filter style is composed on filters (components) and pipes (connectors). This is a more abstract notion of software architecture than ours because both the pipes and filters correspond to components (instances) in a situation architecture. The style itself corresponds to a family of related domain architectures that cover the design space of pipe and filter systems.

The pipe and filter style is thus more specific since it only applies to systems for which interpreting instances and either pipes or filters is advantageous. These advantages lay in the fact that the interaction between pipes and filters is quite straight forward. In turn the uniformity of the relationship allows us to easily abstract it away providing a specialized style in which to understand such software systems.

An similar style would exist whenever there is a small set of different kinds of instances that interact in a simple and straight forward manner.

Layered Style: The layered architectural style is based on layers (components) and dependencies between these layers (connectors). If we compare

1. or perhaps more correctly on message sends

the style with the notion of a domain architecture then we see that sets of classes are grouped into a single component and if any class depends on a class in another group then the layer depends on the layer to which that class belongs.

This is also a more abstract notion of architecture since sets of components are mapped to a single component and sets of connectors are mapped to a single connector. The reason such a view is effective is that many systems consist of various subsystem and there is a strict hierarchy where subsystems only depend on other subsystems that are lower in the hierarchy.

The advantage is that we know that an situation architecture and thus an architecture corresponding to the layered style will posses the same structure. That is, the instance will be separated into layers and will depend only on layers composed of instances lower in the hierarchy.

The justification of such an approach is based on the fact that there is more cohesion within a layer than between layers. The advantage is that it views a software system at a much larger granularity than a situation architecture that allows a better understanding of the system as a whole at the expense of information about a particular layer.

2.2 Summary: Architectural Styles

The relationship between the different styles can perhaps best be understood in terms of an example. If we consider an object-oriented implementation of a pipe and filter system then the domain architecture would be the set of classes and a particular system would be represented as a situation architecture. Simultaneously we could understand the system in terms of the pipe and filter architectural style. If we wish to consider how the pipes implement their communication then the layered style may also be valuable.

Architecture styles are at different levels of abstraction and also differ in their generality; a layered style is more general that the pipe and filter style for example. In this sense architectural styles are a somewhat eclectic mix of known ways of understanding software systems. Some, such as the object-oriented style, are not even architectural in nature.

In general an architectural style is easy to map back to the notions of a domain and situation architecture. The exception to this rule is the object-oriented style where the choice of basing the connectors on method invocation is incompatible with the relationship based approach used to define the notion of a situation architecture as well as the other two styles used as examples.

2.3 Patterns and Frameworks

Much of the early work on software architecture came from outside the OO community where the work on Frameworks and design patterns originates. Consequently the relationship between these notions is often vague and ill defined. In fact Mellor and Johnson [2] go as far as to say that much of what is written by one camp is nonsense unless you understand their underlying assumptions.

Since FAMOOS has the stated goal of evolving object-oriented legacy systems into frameworks and we have concluded that many of the problems uncovered in the case studies are architectural in nature it is only natural to seek to reconcile these different views.

To help make these new concepts more intuitive as well as to show both their applicability and utility we will use them to explain the familiar concepts of design patterns and frameworks.

Neither patterns nor frameworks are purely architectural; both describe interactions at the structural and behavioral level. However, our interest is in architecture so we intentionally limit our focus to the structural relevant details of design patterns and frameworks.

Design patterns [1] have a structure that enables a set of desired interactions to take place. Typically, they are motivated by an example, expressed in terms of instances, and document a general class structure supporting the desired interactions. Structurally, the example used to motivate a design pattern is a fragment of a situation architecture and the solution (the pattern itself) is a fragment from a corresponding domain architecture.

Design patterns relate part of a desired situation model, i.e. a concrete situation expressed in terms of instances, to a corresponding domain architecture supporting the required behavior, i.e. relationships and their evolution. Viewed from the opposite perspective, design patterns document substructures within domain architectures that occur frequently because they solve situations common to many software systems.

A design pattern is thus conceptually on a smaller scale than an individual architecture and numerous design patterns may be found in a single domain architecture. A design pattern is similar to a domain model in the sense that they both provide general solutions to a class of a related problems; however, the major difference is the a pattern is generic and will occur across many otherwise unrelated domain architectures.

Frameworks are an approach to facilitating the rapid development of families of related software systems. A framework itself is close to the concept of a software system and thus consists of both domain and situation architectures.

The difference between a conventional software system and a framework is that a framework is customizable. This is accomplished by identifying locations in the different domain architectures, called hot spots, where they may be easily extended. Related software systems are created by creating new configuration of instances, i.e. situation models after first extending the domain architecture if necessary.

Extending a framework does not (ideally) involve fundamentally altering the core domain architectures. This is because the key structure, possessing the required flexibility, is already present in the domain architecture in anticipation of the needs of future software systems. Consequently, we can view the family of related software systems as sharing a common domain architecture.

In contrast changing the configuration of instances can substantially alter a situation architecture. This is explained by the fact that a domain architecture is based on potential relationships while an situation architecture is based on actual relationships. It is thus the situation architecture that is essentially responsible for the individuality of each software system.

A framework is a larger concept than an individual domain or situation architecture. A major advantage of a good framework is that most if not all of the implementation decisions about how different aspects will be implemented have already been made. This greatly facilitating the production of similar applications.

2.4 Summary: Patterns and Frameworks

Once again, just to make it clear, neither design patterns and frameworks are purely structural and thus cannot be reduced to their relationship with an architecture. However, understanding their relationship to a particular notion of software architecture is important.

This relationship is mostly seen in their different scopes; a design pattern has a smaller scope than an architecture while a framework has a larger scope than either a domain or situation architecture. Other notions of software architecture may also be applicable in particular cases but in general it is not possible to relate general concepts such as design patterns and frameworks to architectural styles for example.

Acknowledgments

This work has been funded by the Swiss Government under Project NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT project 21975

References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vilissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[2] Robert T. Monroe, Andrew Kompanek, Ralph Melton and David Garlan, “Architectural Styles, Design Patterns and Objects,” *IEEE Software*, January 1997, pp.43-52.

[3] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.